CHAPTER

2

EXAMINING THE PARTS OF AN OPERATING SYSTEM

After reading this chapter and completing the exercises, you will be able to:

- Explain the purpose of operating systems
- List the parts of a modern operating system
- Explain how the pieces of an operating system relate to one another
- Describe the difference between kernel mode and user mode and explain why the difference exists
- Relate this information to the Windows 2000 architecture

The basic functions of an operating system are to create an operating environment for applications and to provide those applications with some base functionality. Technically speaking, an operating system isn't strictly necessary. You *could* design an application to interact directly with computer hardware, reading and writing data to disk and displaying it on the monitor. An old version of WordPerfect did something similar, bypassing DOS to pull data from the keyboard buffer so that it could display user input more quickly than could applications that relied on the operating system. Designing such an application would be difficult, however, because it would require application designers to build code to interact with system hardware into the applications. It would also mean that the designers would have to create a different version of the application for every hardware version—something that is very difficult to do. Rather than design applications to interact with system hardware, operating systems now handle that task. Of course, operating systems provide other important functionality.

Another reason to prevent applications from communicating directly with hardware is that separating the two keeps applications from messing up the hardware in a way that crashes the system. Games supply a prime example of this kind of problem. Imagine that you're running around in MechCommander, or some other game, under Windows 98, about to capture the fort . . . and then the music begins droning a single note and the screen freezes. Some kind of communication with the sound card has caused not only the game but also the entire operating system to crash. If you press Ctrl+Alt+Delete to bring up the task list, nothing happens—the mouse and keyboard no longer respond. There's nothing left to do but press the Reset button on the computer and reboot. If the operating system handles all communication between applications and hardware, however, then communication with hardware happens in a standard way and this kind of scenario becomes much less likely.

Some operating systems, such as Windows 9x, permit applications to have some communication with hardware, but Windows 2000 (Win2K) and its Windows NT predecessors do not. For this reason, some games do not run under Windows NT or Win2K—they need to access system hardware to work. The Win2K insistence that it handle all communication between applications and hardware limits the set of applications that will run under the operating system, but also makes Win2K more robust (that is, less likely to crash) than operating systems that do permit such communication.

This chapter discusses the parts of a modern operating system, using Windows 2000 as a model, and then shows you how Win2K organizes those parts into a coherent whole and uses them to support applications and user needs.

General Principles of Modern Operating System Architecture

Traditionally, operating systems were monolithic. All of the **helper** parts of the operating system—the parts that allowed applications to communicate with hardware—were lumped together in a single unit and communicated with each other in a separate area of memory, away from applications. As shown in Figure 2-1, these parts were interdependent, which meant that changing one part—such as the way device drivers worked—meant that you might have to update the rest of the operating system so that it would continue to function properly. Unfortunately, the fact that all parts of the operating system were designed to run in the same memory space meant that one part of the operating system could corrupt another part. For example, if the first part was badly designed, it might do something it wasn't supposed to do, such as editing the data accessible to the second part.

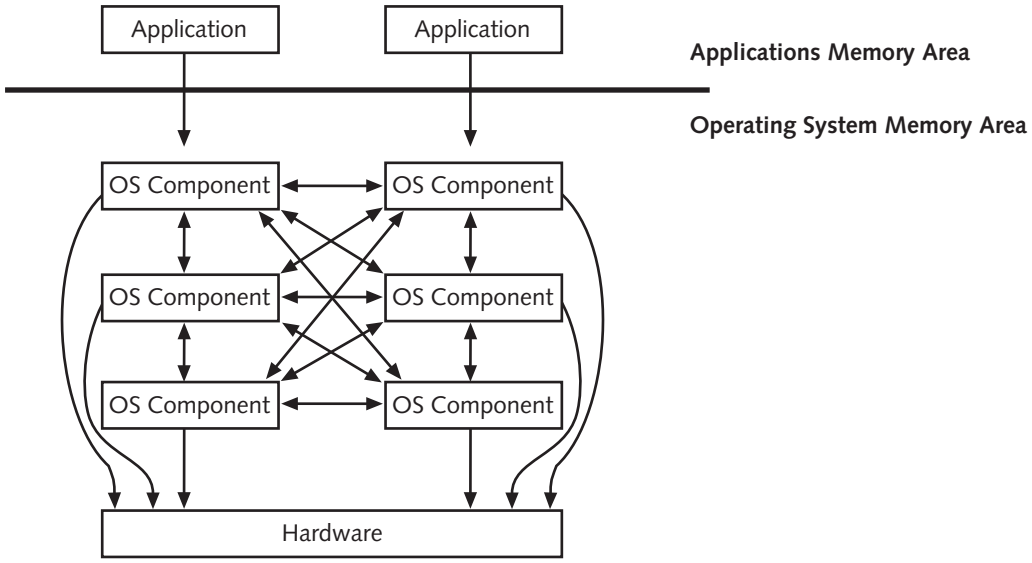

Figure 2-1 A model of a monolithic operating system

Because of these drawbacks, modern operating systems are inclined to take a different approach, using a layered model or a client/server model. In the layered model (see Figure 2-2), the operating system is divided into modules that are layered on top of each other like a wedding cake. Also like a wedding cake, the pieces are interdependent to some degree—if you remove the bottom layer, the layer on top of it will collapse. They are also modular, however. To continue with the wedding cake analogy, if the original operating system were all vanilla and you replaced layer 3 with a chocolate layer, the whole thing would still work together—you wouldn't have to redesign the other layers to work with the chocolate layer. In a layered operating system, each module contains code that the other layers can talk to so as to communicate with that layer or transfer data between the layers.

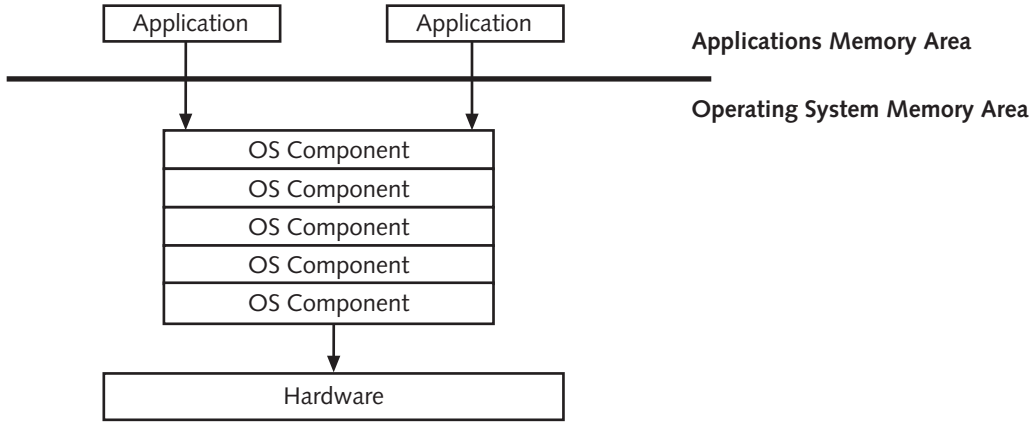

Figure 2-2 A model of a layered operating system

The client/server operating system model (shown in Figure 2-3) divides the operating system into a number of server processes, each of which implements a part of the operating system, such as memory management or accessing hard disks. The **client** (an application or another server process) sends a message to the server process to do something. A messaging part of the operating service passes the message to the server process, which then does whatever the client requested. The messaging part of the operating system then passes the results back to the client. The only disadvantage of the client/server model is that, in its pure form, it's very slow because of the messaging back and forth that is needed to get anything done.

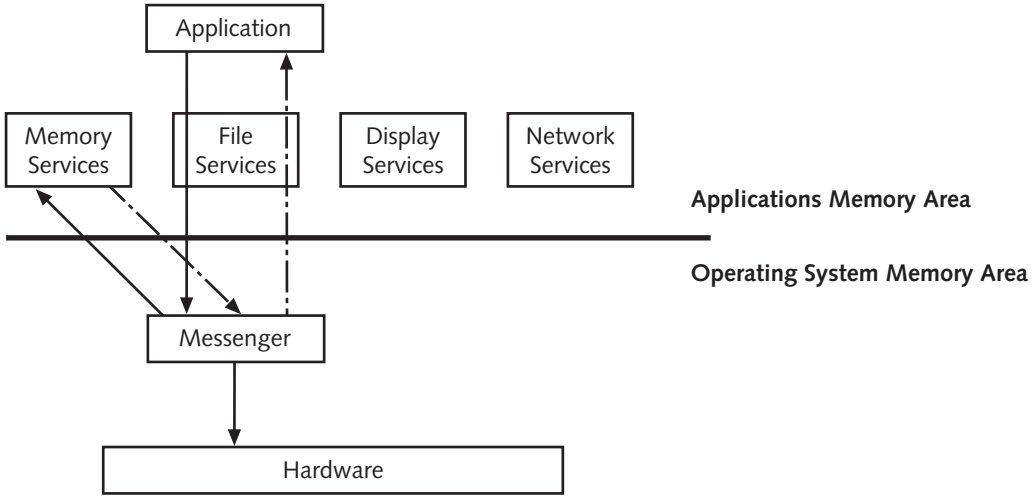

Figure 2-3 A model of a client/server operating system

Windows 2000 combines the layered model and the client/server model. It is designed as a collection of discrete modules that can communicate with each other either directly or via the messaging component mentioned earlier; the modules are layered so that they support each other as well. The remainder of this chapter first discusses the parts of Windows 2000 and then considers how those parts fit together.

Environmental Subsystems

For applications to execute in an operating system, they must have access to those pieces of the operating system that contain the functions they need, such as functions providing for allocation of CPU time or use of RAM for storing data. The part of an operating system that enables such access is called the **environmental subsystem**. Different environmental subsystems expose different parts of the core operating system capabilities. Consequently, applications, which are built to expect a certain kind of support from their operating environment, can run only in that environment. For example, UNIX applications can't run in Windows because the UNIX and Windows operating environments are different. The only way that you can run an application in an operating environment for which it wasn't built is via a mechanism called **emulation**. Emulation creates a compatible operating environment for the application, then

translates the application's requests so that the local operating environment can execute them. Because of the overhead incurred in the translation process, emulation is slow.

Windows 98 supports only a single environmental subsystem, which you see when you start up the computer. Win2K supports three such subsystems: the 32-bit Windows (Win32) subsystem you see when you start the computer, POSIX, and, on x86 systems, OS/2 1.0. (The most recent version of OS/2 is 3.0.) You can review all of the current subsystem information by using the Win2K Registry Editor, which displays a graphical image of all operating system information. As you can see in Figure 2-4, the Win32 subsystem starts by default and is necessary to the operating system, whereas the OS/2 and POSIX subsystems are optional and don't start unless you specifically run them. The "Required" values mean that the Debug and Windows subsystems must be loaded for Win2K to work. Debug is blank because it's used for internal Microsoft testing; Windows is defined as being Csrss.exe, which is the executable name of the Win32 subsystem. To start a subsystem, you must run it like a program, much as you might type Winword.exe at the command prompt to run Microsoft Word. When it's loading, Win2K runs Csrss.exe automatically.

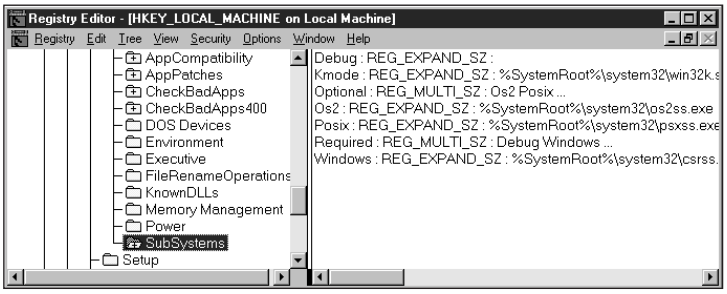

Figure 2-4 Registry values for environmental subsystems

A 32-bit operating system supports 32-bit instructions, which is the technical way of saying that every communication it makes with a computer's hardware contains 32 bits of data. The larger each instruction is, the more capabilities the operating system has, because it can tell the hardware to do more things in a single instruction. Win2K is a 32-bit operating system, as is the Windows environmental subsystem. The OS/2 and POSIX environmental subsystems are 16-bit, which makes them slightly slower than the other subsystems.

How do applications use the subsystems to communicate with the operating system's capabilities? Each subsystem contains a collection of **function calls**, which are predefined messages that the subsystem can use to communicate with the core operating system functions. (They're referred to as "calls" because a program "calls" a function to use it.) A subsystem's communications are limited to those function calls, just as a person who speaks only English is limited to expressing the thoughts for which English words exist.

In Win2K, function calls can't be mixed between subsystems. That's because although two subsystems may share some capabilities, they don't use the same function calls—they don't speak

the same "language." For this reason, an application designed to use a particular set of function calls can't run under a subsystem that doesn't support them. This fact explains why the POSIX and OS/2 environmental subsystems aren't good for much—they're very abbreviated versions of the environmental subsystems and don't support many of the function calls used by other versions of these subsystems. Consequently, you can't even run all OS/2 applications in the OS/2 subsystem because the subsystem doesn't support all of the OS/2 function calls. Not all OS/2 functions are included in the environmental subsystem, although most of the functions required to run and display OS/2 applications are. Thus, unless you must support POSIX or OS/2 applications, use of these subsystems is not recommended.

Win2K has two classes of functions: those that call parts of the core system services, and those that call a support function, such as a part of the Win32 subsystem or the heap service that allocates resources to applications. Both types of function calls are part of **dynamic link libraries (DLLs)**, which are essentially function-specific collections of functions that an environmental subsystem supports. The Win32 subsystem, for example, supports Gdi32.dll, which is the dynamic link library that contains all of the functions needed to support graphical output. Collectively, an environmental subsystem's DLLs that provide a standard interface to the core operating system functions are called its **application programming interface (API)**. That is, the API provides applications with a standard interface to the workings of the computer. It makes it possible to change the inner workings of the operating system without affecting the applications, so long as the function calls remain the same.

A good analogy is driving a car. Because cars have a standard API (steering wheel, clutch, windshield wipers, and so forth), once you learn how to drive a car, you can drive any vehicle that works like the one on which you learned to drive, whether it's a Hyundai or a Porsche. The interface for the cars remains the same even if the inner workings do not. Along these same lines, if an API of an OS has changed, the OS itself remains largely the same, but functions differently under the hood. Consider the Telephony API (TAPI). Because communications carriers' technology changes frequently, the API must change to provide the most efficient method for communication between the carrier and the operating system. Users remain largely unaware of these changes, because they just connect to the Internet in the way they always did through the operating system.

Not all DLLs reside in the environmental subsystems. Ntdll.dll is a collection of two classes of system functions: one to call core operating system functions and one to call functions that are part of subsystems.

Win32 Subsystem

The Win32 subsystem is the most important environmental subsystem for Win2K, because the rest of the operating system depends on it to expose the necessary parts of the core operating system. As noted earlier, the other two subsystems start only on demand, when you run an application that requires their support. The Win32 subsystem, in contrast, must always be running. If the environmental subsystem crashes, or you stop it, then the entire operating system will crash and you must restart the computer.

The Win32 subsystem contains the following features:

- The environmental subsystem process (Csrss.exe), supports display of text windows (such as the Win2K command prompt), provides the ability to create processes and threads, and offers some support for creating the Virtual DOS Machines that Win2K uses to support DOS and Windows 3.x applications.
- The Win32K.sys device driver contains the Windows Manager, which is responsible for displaying window-based applications (which means any Windows application) and collecting user input, and the graphical device interface (GDI), which creates printer and monitor output.
- Subsystem DLLs translate application calls directed to the API to the core operating system components.
- Graphics device drivers take the graphical instructions that Gdi32.dll puts together and pass them to the video and printer drivers that constitute the software interface to the video boards and printers.

If you wanted to run Microsoft Word application, for example, you'd run it in the Win32 subsystem that's already running, because Microsoft Word requires some function calls found only in this operating environment. When you click the Microsoft Word icon to start the application, Csrss.exe starts the application, User.dll draws the application's windows and buttons, and Gdi32.dll sends the windows and buttons to the monitor for display. When you save a document, another one of the subsystem .DLLs calls the operating system's ability to write data to a file on the hard disk.

Win32 applications are not the only ones to use the Win32 subsystem. If you run a Win16 or DOS application on a Win2K machine, when it starts the application, Win2K creates a special operating environment to accommodate the legacy application. This operating environment runs in the Win32 subsystem, rather than directly in the Win2K operating environment.

POSIX Subsystem

The Portable Operating System Interface (based on) UNIX (POSIX) is a collection of international standards for UNIX-like operating system interfaces. The idea is that, if the vendors creating operating systems follow a standard, then applications built for one version of POSIX will work on all versions of POSIX. This standardization is necessary because of the plethora of essentially incompatible UNIX versions that exist.

The Win2K POSIX environmental subsystem is based on one of the many POSIX standards (implying that this idea of creating a standard interface doesn't seem to have worked so well): POSIX.1. Microsoft included POSIX in the original design of Win2K mainly to satisfy a U.S. government requirement for POSIX support. POSIX.1's API is very limited, however, and requires some support from the Win32 subsystem for display and communication between applications.

OS/2 Subsystem

The OS/2 environmental subsystem is based on an old version of OS/2— that is, version 1.2. Like the POSIX subsystem, its capabilities are limited even for those who would like to run OS/2 applications. This subsystem is supported only on x86 systems, not the Alpha chips used in high-end servers. More important, it supports only OS/2 1.2 character-based applications; it cannot handle the graphical OS/2 applications. (An add-on to the subsystem can display the OS/2 2.1 graphical interface, but the OS/2 API still supports only character-based applications.) Even those applications are limited in their functionality, however. As noted earlier, for security reasons, Windows NT-based operating systems don't allow applications to directly access hardware, so OS/2 applications that use a technique called advanced video (which directly communicates with the video card) aren't supported in the Win2K environment. Like POSIX, the OS/2 environmental subsystem is largely a decorative add-on to Win2K, rather than a useful component.

Executive

The executive portion of Win2K is loaded with a file called Ntoskrnl.exe, which you can find in the *%systemroot%*\system32 folder (WINNT\system32 on most computers). The executive component of the operating system is modular by nature and contains the following major items:

- Process and Thread Manager
- Virtual Memory Manager
- Security Reference Monitor
- Input/Output (normally shortened to I/O) Manager
- Cache Manager

The following sections discuss each of these modules in detail.

Process and Thread Manager

The Process and Thread Manager creates and terminates **processes** and **threads**. Processes and threads are discussed further in Chapter 4, "Allocating and Managing Operating System Resources." The basic idea is that the executable files on your system—files such as Sol.exe—don't actually do anything. These files are known as **executable images**. When you start an executable image, Win2K first determines what kind of file it is and what kind of environmental subsystem it requires. The Process and Thread Manager then creates a process for that image; this process defines the relative importance of the functions that the executable is supposed to carry out, gives it some virtual memory areas in which to store data, and defines the operating environment available to the image.

Typically, but not always, the ratio of processes to executable images is 1:1.

One part of the process is called *a thread*, which is the executable element of the image. Every process always has at least one thread, though it may spawn more to support the application as necessary. Threads are allocated CPU time based on their relative importance and are the "worker bees" that allow the executable file to carry out its intended tasks.

As each thread finishes, it terminates and releases its resources back to the process pool. When the last thread in a process terminates, the Process and Thread Manager ends the process and releases all of the process's resources so that other processes can allocate them to their threads.

Although the processes themselves don't execute, people—and the Win2K diagnostic tools—often refer to processes running on the server, rather than threads. It's a convenient form of shorthand that refers to all threads in a process.

Win2K has another process-related structure that's new to the Windows NT environment: **jobs**. A job controls certain attributes of the processes associated with it, such as the default working set (the amount of process-related data that the process gets to keep in RAM, rather than have paged to disk; the next section provides a brief introduction to virtual memory) allowed by each process within the job, its total CPU time limit, the per-process CPU time limit, the maximum number of processes associated with the job, the priority class for the processes, and the processor affinity, if any (that is, the preferred processor to use in a multiprocessor computer). The main function of a job is to allow Win2K to deal with certain processes as groups, rather than as separate tasks. This functionality is new to Win2K, and few applications currently take advantage of it.

Virtual Memory Manager

One of the resources that a process makes available to its threads is a set of virtual memory addresses. **Virtual memory** is a method of using both physical memory (RAM) and a paging file stored on the hard disk for data storage. Data that processes are currently using is stored in RAM where the processes can access it quickly and is called the processes' **working set**; data that the processes have used but are not currently manipulating is stored in the paging file.

Storage of data by the Virtual Memory Manager can be compared to clothing storage. Data in RAM is analogous to the clothes in your closet—the things that are currently in use or frequently accessed. Data in a paging file (stored on the hard disk) is analogous to the clothes you store in your attic—not in current use or used less frequently. The Virtual Memory Manager stores less frequently used data on disk in a paging file and more frequently used data in RAM. Just as it takes longer to retrieve clothing from the attic than from your closet, it takes longer to pull data from the paging file (milliseconds, or thousandths of a second) than from RAM (nanoseconds, or billionths of a second). The advantage of using an attic (or paging file) is that it provides more storage space.

Each process running on a Win2K computer has 4 GB (gigabytes—that's billions of bytes) of virtual memory that it can see. Of this memory, 2 GB consist of system memory addresses, which core Win2K functions use to store data and which are common to all processes running on the system. The other 2 GB are reserved for the use of the threads running in that particular process. This statement doesn't mean that the process has 2 GB of RAM reserved

for it—even the most heavily loaded Win2K servers max out at 1 GB of RAM. Instead, it means that the process can see 2 GB of personal addressable memory spaces in which to store data. Other processes will also see 2 GB of personal addressable memory spaces. One of the jobs of the Virtual Memory Manager is to keep straight how these virtual memory addresses map to physical memory—RAM and the paging file—and to retrieve data requested by processes. Chapter 4 discusses memory management in more detail.

Security Reference Monitor

The Security Reference Monitor is responsible for enforcing the security settings defined in the security subsystem. Security settings in Win2K are a matter of rights (what actions you can perform) and permissions (what objects you can access). Each user or predefined group of users has a **security ID (SID)** associated with it. Chapter 12, "Securing Network Resources," discusses Win2K security systems in detail. For now, suffice it to say, that, when you log onto the domain, the operating system evaluates your SID and gives you an access token based on your SID, which is in turn determined by the security restrictions of the user group to which you belong and any settings that the administrator has applied directly to your account. The access token acts like a pass; it details the things that you're allowed to do on the network.

Each object on the network (each file, application, shared resource, and so on) also has an **access control list (ACL)** that defines the permissions or rights needed to use that resource and specifies precisely how each permission set allows that object to be used. For example, the file object Mydoc.doc could have an ACL that says, "Members of the Users group can read this document, members of the Administrators group can read and edit this document, and members of the Guests group can't see it, let alone read or edit it."

When you try to access an object on the Win2K domain, the Security Reference Monitor compares your access token to the information in the object's ACL. Based on the results of the comparison, you're either permitted a degree of control over that object or told that you're not allowed to use it. If auditing is turned on, then the Security Reference Monitor not only will decide who gets access to what, but will also record successful and failed attempts to access objects or change security settings—and record who instigated those attempts.

I/O Manager

I/O covers how data gets into the computer and how it gets out. The I/O Manager is the piece of Win2K that implements all I/O—no matter what kinds of devices are involved (printers, hard disks, microphones, keyboards)—and communicates with the device drivers that handle the actual communication between the operating system and the hardware. Because I/O is a major part of any operating system, it is discussed in many places throughout this book.

Cache Manager

The Cache Manager is in charge of the **file system cache**, which is a range of virtual memory addresses reserved for storing recently used data related to file sharing from any storage medium (hard disk, CD-ROM, network-accessible drives). This cache holds data related to

file reads and writes—not only the file contents themselves, but also a catalog of disk structure and organization (so file-based I/O is faster). In addition, it holds disk updates in RAM for a brief period, writing them to disk when the system is less busy. The Cache Manager oversees all of these operations, telling the I/O Manager when to retrieve recently used data from the file system cache instead of from the hard disk and ensuring that the contents of the cache are written to the hard disk periodically so that they will not be lost if the server crashes. Because it must shuffle data between RAM and the hard disk, the Cache Manager relies on the Virtual Memory Manager to help it with these tasks.

Other Executive Support Functions

Besides these main groups, the executive part of Win2K includes some functions that support them and, in some cases, work with device drivers. The two most important are the **Object Manager** and the **local procedure call (LPC) facility**. The Object Manager creates the objects that represent executive-level operating system structures such as processes and threads (which are then managed by the Process and Thread Manager). The LPC facility handles interprocess communications—messaging between client and server processes on the local computer (this messaging piece allows the parts of a client/server operating system to communicate). Other functions handle the basic processing required to convert data from one format to another, to organize the security structure, and to allocate memory to Win2K core functionality.

The Kernel

The most important part of Win2K is the kernel, which is the other half of Ntoskrnl.exe and the part that juggles all of the low-level scheduling required in the operating system. The threads that the Process and Thread Manager creates are scheduled for CPU cycles by the kernel. Likewise, the interrupts that I/O devices use to tell the CPU that they have data to process are handled by the kernel. When a process asks the Virtual Memory Manager to retrieve a piece of data that's been swapped out to the paging file, the kernel provides the core functionality that allows the Virtual Memory Manager to realize what it must do to transfer the data to the process. The kernel doesn't make policy—that's up to the subsystem and executive parts of Win2K that have already been discussed. It also doesn't do any error checking to verify that a subsystem is allowed to do what it's trying to do. Rather, it carries out the support functions that allow the executive and the environmental subsystems to operate.

Because the kernel is so crucial to Win2K, it's different from other parts of the operating system. Most of the operating system has pieces that can be paged in and out of RAM, being retrieved from the paging file when they're needed. The kernel is different—it's *always* stored in RAM or, as it is commonly referred to, *resident in memory*. Kernel threads have a higher priority than other threads; therefore, although it will relinquish CPU time so that the CPU can handle interrupts, the kernel won't be interrupted in any of its functions to let other threads run.

Another difference between the kernel and the rest of Win2K relates to the kernel's simplicity. The kernel is designed to do its job as quickly as possible. For this reason, it must leave all policymaking to the other parts of the operating system. If a user wants to run an application, the

kernel isn't concerned with who the user is or whether that person has the right to run that application. It just creates the threads, schedules CPU time for them, and assumes that the Security Reference Monitor has already checked the permissions.

Broadly speaking, the kernel has two main goals: to provide control objects that carry out the functions ordered by the subsystems and executive, and to create a uniform interface that the executive and system device drivers can use to interact with the hardware architectures (meaning x86-based computers or Alpha-based computers).

The Puppet Masters: Kernel Objects

Why does the kernel need to create objects (such as threads) if a part of the executive is already creating them? The answer is, because the executive isn't making them, or isn't making them complete and able to control themselves. The Process and Thread Manager issues instructions to create a thread. Those instructions, in turn, are passed to the kernel to execute. The executive part of the operating system represents threads and other shareable resources as objects. Objects have attributes, such as security settings (ACLs), resource quotas that must reflect the amount of resources used by the object, and handles that allow other objects to manipulate them. All in all, an object represents a formidable amount of bookkeeping. To make Win2K more responsive, the kernel manipulates these complicated objects with simpler **kernel objects** that contain no security information or other attributes, but nevertheless help the operating system organize and time the execution of the executive-level objects. Most executive level objects are linked to one or more kernel objects that manipulate the executive-level objects.

There are two kinds of kernel objects. One type, called **control objects**, controls various operating system functions, such as running the kernel process (recall that the executable image Ntoskrnl.exe that supports the kernel and the executive doesn't do anything; it's just a framework for processes and the threads inside them) and handling hardware interrupts. The second type, dispatcher objects, synchronizes events on the computer and affects thread scheduling. Chapter 4 covers more details about how control objects and dispatcher objects make Win2K work.

Kernel-Level Hardware Support

The second major job of the kernel is to keep the executive and subsystems from having to worry about the kind of hardware on which they're running. To achieve this goal, the kernel has to smooth out any variations in the way that different architectures handle such events as hardware interrupts and page faults. If the kernel didn't take care of this job, then variations in the hardware platform would create similar variations in the executive and subsystems. That said, the kernel itself doesn't really change much to reflect different hardware. It will vary occasionally depending on the hardware installed in the computer on which you install Win2K. For example, the kernel for an Alpha-based machine is different from the kernel for an x86-based machine, because the CPU cache in an Alpha chip works differently from the one in an x86 chip. Nevertheless, the kernel is designed to make it as portable across different hardware as is possible.

The kernel does not interact directly with computer hardware or provide a consistent interface between the operating system and the hardware. That's the job of the hardware abstraction layer (HAL), discussed in the next section. The hardware interactions that are truly hardware-specific and can't be smoothed over with the kernel are included in the HAL.

Hardware Abstraction Layer

When Windows NT was first designed, a crucial element of its design was portability, or the ability to be used on more than one hardware platform. Not all operating systems are portable, and some are more portable than others. In an earlier version, Windows NT was actually more portable than it is now. Windows NT 4 would run on x86-based computers, Alpha-based computers, and computers based on the MIPS and PowerPC chips. Microsoft dropped support for the latter two platforms when the MIPS chip was discontinued and the PowerPC version of Windows NT didn't sell. Win2K runs on only the Alpha and x86 platforms.

The hardware abstraction layer is a key component to making Win2K portable. This loadable kernel-level DLL (called Hal.dll; stored in the *%systemroot%*\system32 folder on a Win2K computer) provides the low-level interface to the hardware of the computer on which Win2K is installed. It hides all hardware-dependent details, such as I/O interfaces, interrupt controllers, and support for multiprocessor computers, or any mechanism that varies significantly with the hardware in the computer. Win2K supports many HALs, but installs only the one you need for your hardware platform. In some cases, the computer manufacturer must provide the HAL (for example, for computers that have more than four processors).

To keep from having to provide numerous versions of the executive component of Win2K, Microsoft maintained portability in Win2K by having the executive call HAL routines when they are needed to access hardware. This approach is roughly equivalent to the way that applications call on the Win2K environmental subsystem DLLs to access core Win2K functionality. The HAL is Win2K's API to the hardware and, as such, the only part of Win2K that interacts directly with system hardware. Even device drivers must call on the HAL.

Because they must interact with hardware directly, HALs are written in a low-level programming language called assembler, rather than the higher-level C in which the rest of Win2K is written. Assembler language code executes more quickly than C code because it has less overhead associated with it. Writing C code is much simpler, however. Although the syntax for assembler language is cryptic, with a little practice, you can read C code to see what it's doing even if you can't program in it.

Device Drivers

Device drivers are modules in the kernel that act as go-betweens for the I/O subsystem and the HAL. There are several types of device drivers:

- **Hardware device drivers**, such as printer drivers, write data to or retrieve it from a physical device or network, manipulating the hardware via the HAL. These are the device drivers you probably think of first.

- **File system drivers** are how Win2K translates file-oriented I/O requests and communicates them to a hard disk or other storage media, like a tape drive or CD-ROM.
- **Filter drivers** intercept I/O requests and perform some processing on them to make the requests intelligible to the receiving devices. For example, if you've logically merged two physical disks into a single volume, any I/O requests to that disk must be translated so that the HAL knows with which physical disk it must communicate. Some quota management software is also based on filter drivers, because the quota system must see the available capacity of the hard disk before permitting anyone to write to it.
- **Network redirectors and servers** are file system drivers that transfer data to and from network-accessible drives. In fact, when a Win2K computer is set up for networking, the redirector intercepts all file read or write requests and examines them to see whether they need the redirector's help. If the I/O request is intended for a locally accessible hard drive, then the redirector relinquishes the request to the file system driver.

Win2K loads device drivers when you boot up the computer, but the drivers don't actually do anything until they're called. If a thread initiates an I/O function (such as a request to retrieve a file from disk) or a piece of hardware interrupts the CPU with a request to process some data, the appropriate device driver communicates with the HAL to retrieve the necessary data.

Outline of Win2K Architecture

To really understand how the pieces of Win2K interoperate, you need to see how they fit together. This chapter has discussed some of the communication that takes place between some parts of the operating system (like the device drivers depending on the HAL to actually communicate with hardware, rather than doing it themselves). The following sections will describe the organization of the Win2K pieces in more detail.

User Space and Kernel Space

Before you can understand how Win2K is organized, you must recognize the difference between user space and kernel space. To prevent user applications from accessing or corrupting data that Win2K needs, Win2K uses two processor access modes: **kernel mode** and **user mode**. User application code runs in user mode; operating system code, such as device drivers, runs in kernel mode.

The basic difference between the two modes is the degree of access that processes are granted to the system. Kernel-mode processes (that is, processes that run in kernel mode) can use any CPU instruction and can read or write on any part of system memory (that is, the 2 GB of virtual addresses shared among all processes). User-mode processes, on the other hand, can use only a subset of CPU instructions and can read and write only to the 2 GB of virtual memory addresses private to them. The two modes give Win2K the power it needs yet prevent user applications from crashing the operating system. Although an application running

in one of the environmental subsystems can still crash itself, it won't take down the rest of the operating system with it.

Kernel mode is sometimes referred to as *privileged mode* because of the privileged position occupied by any code executing in that mode.

Virtual memory is divided into pages (sections that an application can use to store data). Win2K tags each page to indicate what access mode the processor must be in to read and/or write to that page. Threads can access pages in system space (with addresses ranging from 800000000 to FFFFFFFF) only when they're running in kernel mode. Threads running in user mode can access only the pages in user space (with addresses ranging from 00000000 to 7FFFFFFF). Figure 2-5 illustrates this distinction.

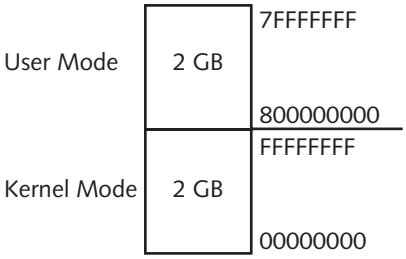

Figure 2-5 Kernel-mode and user-mode virtual memory addresses

The numbering employed by operating systems typically takes the form of a base-16 numbering system called hexadecimal (or just "hex"). The decimal system to which you're accustomed uses one or more of 10 digits to represent any number: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Hexadecimal uses those digits as well, but adds a few more: . . .9, A, B, C, D, E, F. Thus 15 decimal is F hexadecimal. Why use hexadecimal? It's a really convenient way to translate from the binary numbering system that computers read and, with a little practice, hex notation is much easier to read than binary notation.

Threads of user applications switch from user mode to kernel mode depending on what they're doing, using privileged mode when they need to call a system service. For example, if you're running Microsoft Word and try to open a file, the thread responsible for opening that file must call the executive routine that handles file opening. This routine runs in kernel mode. Word itself continues to run in user mode, but gives some time to a system process so that the executive routine will open the file you requested. When the system service is finished, the CPU switches back into user mode (a move called a **context switch**, which is discussed more in Chapter 4) and then gives control back to the Word thread. Thus Win2K never runs in kernel mode when the user application has control.

That's the difference between user mode and kernel mode. Figure 2-6 indicates which parts of Win2K are in each mode.

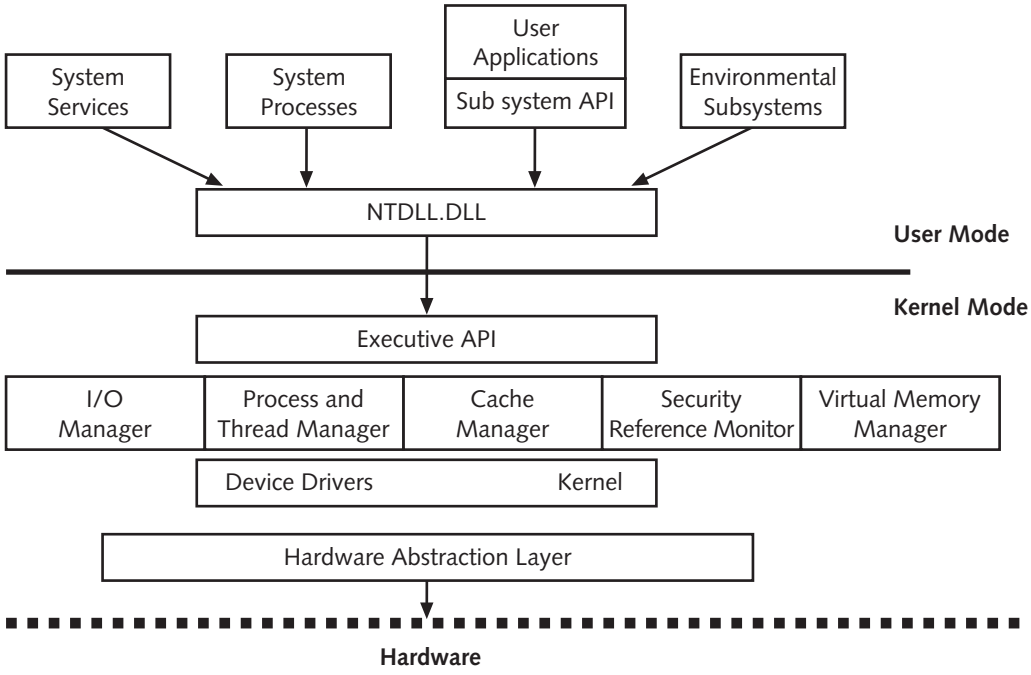

Figure 2-6 A simplified Win2K structure

A thread supporting a user application normally spends some time running in kernel mode and some time running in user mode, depending on what it's doing. For example, if a thread is waiting for user input, it runs in user mode. If the thread is actually performing a computational function, it operates in kernel mode.

Pieces Found in Each Space and Their Relationship to Each Other

The user mode/kernel mode design has several implications. User applications, environmental subsystems, and Win2K services run in user mode and do not communicate directly with the executive, which runs in kernel mode. Rather, the subsystem DLLs—the API—pass information to the executive part. System processes, such as the logon process that passes your user name and password to the security subsystem and that determines whether you're allowed to log onto the computer, run in user mode but can pass information to the executive services without relying on the subsystem DLLs. The kernel and device drivers support the executive, and the HAL performs all communication with hardware that Win2K needs.

The graphics services in kernel mode are actually part of the Win32 subsystem, Microsoft puts them in kernel mode, however, to make Win2K draw windows more quickly.

Chapter Summary

- This chapter introduced the general principles of operating system architecture with a focus on the Windows 2000 (Win 2K) architecture.
- The Windows 2000 virtual memory model combines the use of physical RAM and paging files into a demand paging mechanism so as to maximize memory use and efficiency. Windows 2000 is easy to use, offers new storage capabilities, provides improved Internet access, and maintains strict security.
- Windows 2000 is based on a modular programming technique. Its main processing mechanism is divided into two modes: user mode and kernel mode. User mode hosts all user processes and accesses resources via the executive.
- The separation of modes provides for a more stable and secure computing environment. It supports the application subsystems that enable Windows 2000 to execute DOS, Win16, Win32, POSIX, and OS/2 software.
- The kernel mode hosts all system processes and mediates all resource access. Its executive manages operations such as I/O, security, memory, processes, file systems, objects, and graphical devices.

Key Terms

access control list (ACL) — The list that defines the permissions or rights needed to use a system resource and specifies precisely how each permission set allows that object to be used.

application programming interface (API) — The entire set of DLLs that an environmental subsystem supports to request kernel-mode services.

client — The computer or user that requests information from a server.

context switch — The action that takes place when a processor switches from kernel mode to user mode.

control object — A kernel object that controls various operating system functions, such as running the kernel process.

device driver — A kernel-mode module that acts as a go-between for the I/O subsystem and the hardware abstraction layer.

dynamic link library (DLL) — A specific set of function calls that allows executable routines to be stored as files and to be loaded only when needed by a program that calls them.

emulation — A mechanism by which an environmental subsystem supports applications for which it doesn't have an API.

environmental subsystem — The part of an operating system that provides an interface to the functions that an application needs to support user requests. Win2K supports three environmental subsystems: Win32, POSIX 1.0a, and OS/2 1.0.

executable image — The name of an application or a logical construct for the processes and threads that actually execute the application.

file system cache — A range of virtual memory addresses reserved for storing recently used data related to storage I/O.

file system driver — A device driver that translates file-oriented I/O requests for the hardware abstract layer to pass to storage media.

filter driver — A device driver that intercepts file I/O requests and processes the request to make it intelligible to the receiving device.

function call — A predefined request for a kernel-mode action that the environmental subsystem can call at the request of an application.

hardware device driver — A module that writes data to or retrieves data from a physical device or network, manipulating the hardware via the hardware abstraction layer.

helper — Parts of the operating system that allow applications to communicate with hardware. Originally, these parts were lumped together in a single unit and communicated with each other in a separate area of memory, away from applications.

job — A collection of processes with certain common characteristics, such as the working set and the amount of CPU time that the threads in the process get.

kernel mode — A processing mode that gives complete access to all writable addresses in the system process area. Kernel objects run in kernel mode. Because this mode allows access to the operating system, only code that must interact with the operating system directly runs in kernel mode.

kernel object — An object that exists only in kernel mode and with which the kernel manipulates executive-level objects such as processes and threads. Kernel objects contain no security information or other attributes, so they don't incur the same kind of policy-based overhead that executive objects do.

local procedure call (LPC) facility — The Win2K messaging mechanism that allows client and server processes to communicate.

network redirectors and servers — File system drivers that transfer data to and from network-accessible drives.

object manager — The part of the executive that creates the objects representing executive-level structures such as processes and threads.

process — The context for an executable thread. Processes define the priority, available resources, virtual memory settings, and other settings for the threads executing in the context of that process.

security ID (SID) — The unique identifier that is determined by the security restrictions of the user group to which you belong and any settings that the administrator has applied directly to your account.

thread — An entity within a process for which Win2K schedules CPU time to execute a function of some kind. When a thread has finished its job, it terminates.

user mode — A restricted kind of access to CPU functions and virtual memory. User mode limits user applications to using per-process virtual memory addresses and a subset of CPU functions, allowing them to request kernel-mode functions but not to read or write data in system areas.

virtual memory — A method of using both hard disk space and physical RAM to make it appear as though a computer has as much as 4 GB of RAM.

working set — Data that the thread in a process is currently using and that is stored in RAM.

Review Questions

1. No modern operating systems permit applications to directly access system hardware. True or False?
2. The collection of core system operations on which an environmental subsystem can call are individually known as _____ and collectively known as the environmental subsystem's _____.
3. Which of the following is a group of related function calls?
   a. Dynamic link library
   b. Application programming interface
   c. Environmental subsystem
   d. Operating environment
4. Which environmental subsystem(s) is (are) loaded when Win2K starts up?
5. The OS/2 subsystem is supported only on Alpha Win2K systems. True or False?
6. With which file is the executive portion of Win2K loaded?
   a. Ntldr
   b. Ntdll.dll
   c. Ntoskrnl.exe
   d. Both A and B
7. Which of the following is not part of the Win2K executive?
   a. Win32 subsystem
   b. I/O Manager
   c. Process and Thread Manager
   d. Ntdll.dll
8. The _____ is responsible for storing recently used file data in memory.
9. Every thread has at least one process, but it may spawn more processes to support the actions of the person using an application. True or False?
10. What is a Win2K job?
11. A process's _____ is the set of data that the process currently has stored in physical memory.
12. Memory access times are typically measured in _____; disk access times are measured in _____.
13. Win2K supports _____ of virtual memory addresses.
    a. 2 GB
    b. 3 GB
    c. 4 GB
    d. As much as you install in the computer (typically a maximum of 1 GB in modern computers)

14. How are ACLs used in the Win2K executive? Which part of the executive uses them?
15. Users get _____ based on their security identifiers (SIDs).
16. Which part of the Win2K executive manages the problem of how data gets into the computer and how it gets out?
    a. Security Reference Monitor
    b. Cache Manager
    c. I/O Manager
    d. Device drivers
17. In Win2K, the part of the executive that handles communication between client and server processes on the same computer is called the _____.
18. The Process and Thread Manager creates its own processes and threads. True or False?
19. Which two components of Win2K does Ntoskrnl.exe load?
20. The _____ handles all low-level scheduling for Win2K.
    a. Executive
    b. Kernel
    c. Process and Thread Manager
    d. None of the above
21. The scheduling component of Win2K is always resident in memory. True or False?
22. Which part of Win2K interacts directly with system hardware?
    a. Device drivers
    b. Executive
    c. Kernel
    d. None of the above
    e. Hardware abstraction layer (HAL)
23. List the platforms on which Win2K runs, and compare them with the platforms on which Windows NT 4 ran.
24. _____ are the parts of the kernel that act as go-betweens for the I/O subsystem and the part of Win2K that interacts with hardware.
25. A network redirector is a file system driver. True or False?
26. On what kind of device drivers might quota management software rely?
    a. Filter drivers
    b. File system drivers
    c. I/O drivers
    d. None of the above
27. In Win2K, applications execute in _____ mode.
28. How do user-mode and kernel-mode processes use memory differently?

29. Device drivers run in _____ mode; the executive runs in _____ mode.
30. What is it called when the CPU changes from handling kernel-mode processes to handling user-mode processes?
    a. Context switch
    b. Working set
    c. Paging operation
    d. Local procedure call

HANDS-ON PROJECTS

Project 2-1

The Win32 subsystem is generated when Win2K runs the executable image Csrss.exe at startup time. Win2K depends on the Win32 subsystem to provide basic functionality to the operating system. In this demand, the Win32 subsystem is different from the OS/2 and POSIX subsystems, which are loaded only if you run an application that requires them.

To manipulate any process, you need to know its image name or its process ID (PID; the number identifying it). You already know the image name of the Win32 subsystem; you will find its PID with a tool called the Task Manager.

Many tools will supply you with a PID, not just the Task Manager. Because the Task Manager will be referred to again in the course of this book, you should become familiar with it now.

To find the process ID of the Win32 subsystem:

1. To run the Task Manager, press **Ctrl+Alt+Delete** to open the Windows Security dialog box. You'll have a choice of actions: log off the system, shut down, lock the computer, change the password, and open the Task Manager. Click the **Task Manager** button to open the Task Manager.
2. Click the **Process** tab of the Task Manager, as shown in Figure 2-7.

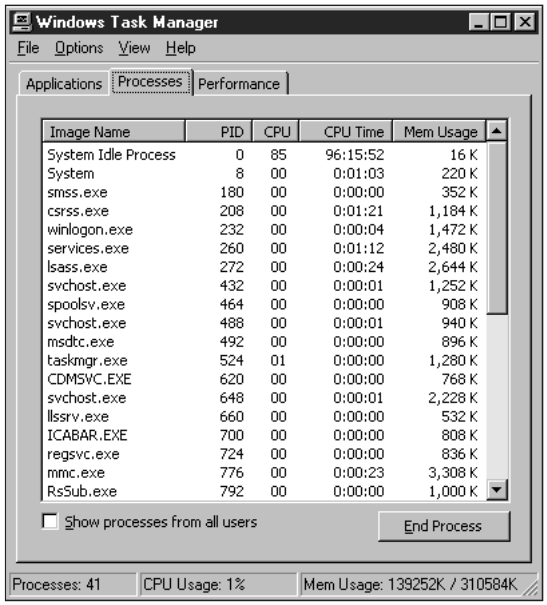

Figure 2-7 List of processes running on the computer

3. Find the **Csrss.exe** image in the list of running processes. When you find it, look at the number opposite it in the PID column. Write down this PID for your instance of the Win32 subsystem—you'll need it for the next exercise.

Project 2-2

If you could stop Csrss.exe from running, you'd crash the entire operating system. In Windows NT 4, you could stop the Win32 process, which would cause a "blue screen of death" and require you to restart the computer. Win2K protects itself from crashing. To see what happens when you try to crash it, follow these steps:

To observe the importance of the Win32 subsystem:

1. Open the **Win2K Resource Kit**, which is in the **Programs** section of the **Start** menu.
2. In the Resource Kit, click the folder for **Tools A to Z**, so that the screen looks like the one in Figure 2-8.

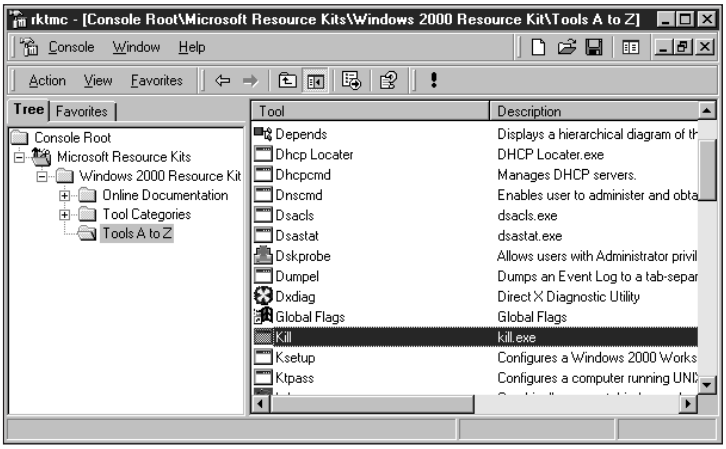

Figure 2-8 The Kill tool in the Resource Kit

3. Double-click the **Kill** tool. You'll open a command-line window that looks like the one in Figure 2-9.

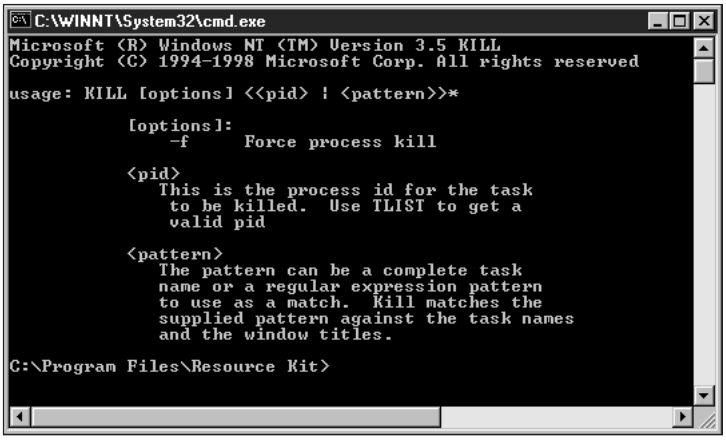

Figure 2-9 Command-line options for the Kill utility

4. Type **KILL *PID***, where *PID* is the process ID you wrote down in Hands-on Project 2-1.
5. Note that you receive the following message:

```
process csrss.exe (204) - '' could not be killed
```

Even if you add the –f switch to KILL, which should force a process to end, you'll get the same message.

Project 2-3

You can use the Win2K Performance Monitor to observe many things about your computer, including the amount of time it spends executing user code versus kernel code. To see this difference, follow these steps.

To find out how much time your computer spends in kernel mode and in user mode:

1. Click Start, Run, and type **perfmon**. When you see a dialog box indicating that the Performance Monitor has been replaced by the System Monitor and telling you to click OK to start the System Monitor or Cancel to start the Performance Monitor, click **Cancel**. (The System Monitor includes support for the Performance Monitor, but keep the interface simple for now.) You'll see a screen like the one in Figure 2-10.

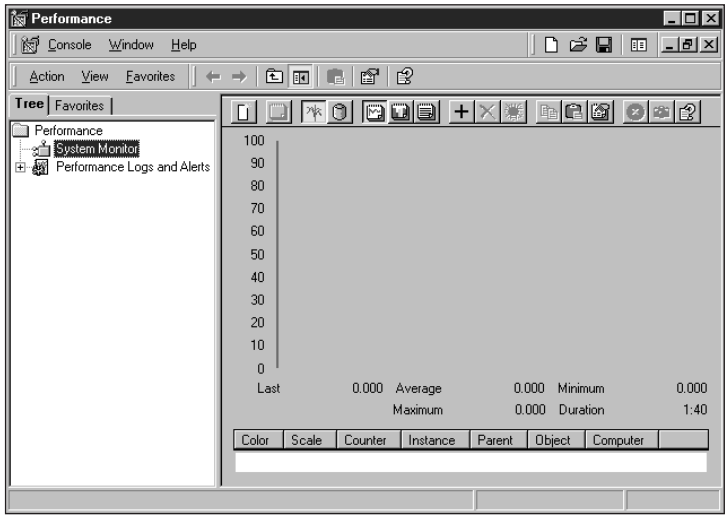

Figure 2-10 Performance Monitor

2. Click the button that has the plus sign (called the Add button) on it to open the dialog box shown in Figure 2-11 (You can also right click the chart and select Add Counters from the resulting menu.)

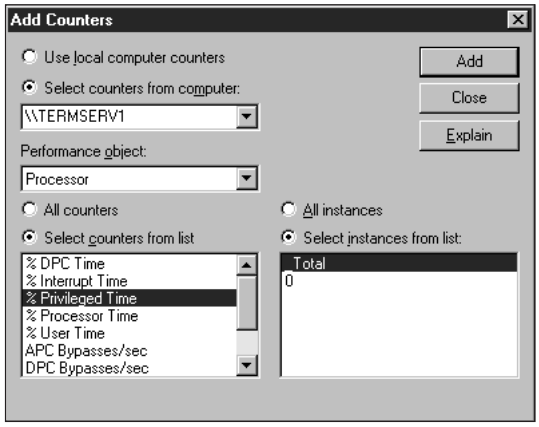

Figure 2-11 Add Counters dialog box

3. In the Performance object field, select the **Processor** object. It should already be visible.
4. In the Select counters from list field, choose **% Privileged Time**. Click **Add**. Choose **% User Time**. Click **Add**.
5. Click **Close** to close the Add Counters dialog box.

A key at the bottom of the Performance Monitor shows you the colors used for each mode of execution. When you do anything on the computer—save a file, move the mouse, open an application—you'll see both chart lines spike. On average, however, the computer spends more time in privileged mode (recall, that's another name for kernel mode) than in user mode.

Case Projects

1. You're installing Win2K on one new computer with 128 MB of RAM and on a second computer with 256 MB of RAM. Otherwise, the two computers are identical. How will the virtual memory address range available differ on these two computers, and why? Will this amount change if you install more memory on one of the computers without reinstalling the operating system?
2. You've installed a new word processor and a new network card driver on your computer on the same day. A few minutes later, when you open a file stored on a network-accessible drive, Win2K crashes. Which of the two new items on your computer is more likely to be responsible for the crash, and (specifically) why?